Efficient Data Integrity Checking for Untrusted Database Systems

Anderson Luiz Silvério and Ricardo Felipe Custódio
Laboratório de Segurança em Computação
Universidade Federal de Santa Catarina
Florianópolis, Brazil
Email: anderson.luiz@inf.ufsc.br
custodio@inf.ufsc.br

Marcelo Carlomagno Carlos
Information Security Group
Royal Holloway University of London
Egham, Surrey, TW20 0EX, UK
Email: marcelo.carlos.2009@rhul.ac.uk

Ronaldo dos Santos Mello
Grupo de Banco de Dados
Universidade Federal de Santa Catarina
Florianópolis, Brazil
Email: ronaldo@inf.ufsc.br

***Abstract*—Unauthorized changes on database contents can result in significant losses for organizations and individuals. This brings the need for mechanisms capable of assuring the integrity of stored data. Existing solutions either make use of costly cryptographic functions, with great impact on performance, or require the modification of the database engine. Modifying the database engine may be infeasible in real world environments, especially for systems already deployed. In this paper, we propose a technique that uses low cost cryptographic functions and is independent of the database engine. Our approach allows for the detection of malicious data update, insertion and deletion operations. This is achieved by the insertion of a small amount of protection data in the database. The protection data is calculated by the data owner using Message Authentication Codes. In addition, our experiments have shown that the overhead of calculating and storing the protection data is lower than previous work.**

Keywords–Data Integrity; Outsourced Data; Untrusted Database; Data Security.

I. INTRODUCTION

Database security has been studied extensively by both the database and cryptographic communities. In recent years, some schemes have been proposed to check the integrity of the data, that is, to check if the data has not been modified, inserted or deleted by an unauthorised user or process. These schemas often try to solve one of the following aspects of the data [1], [2]:

- *Correctness*: From the viewpoint of data integrity, correctness means that the data has not been tampered with.
- *Completeness*: When a client poses a query to the database server it is returned a set of tuples that satisfies the query. The completeness aspect of the integrity means that all tuples that satisfy the posed query are returned by the server.

Trying to assure data integrity, many techniques have been proposed [3], [4], [5], [6]. However, most of them rely on techniques that require modification of the database kernel or the development of new database management systems. Such requirements make the utilization of the integrity assurance mechanisms in real-world scenarios difficult. This effort becomes more evident when we consider adding integrity protection to already deployed database systems.

Most of the remaining work uses authenticated structures [7], [8], [9], based on Merkle Hash Trees (MHT) [10] or Skip-Lists [11]. These works are most simpler to put in practice, since they don't require modifications to the kernel of the Database Management System (DBMS). However, the use of authenticated structures limits its use to static databases. Authenticated structures are not efficient in dynamic databases because for each update the structure must be recalculated.

In this paper, we address the problem of ensuring data integrity and authenticity in outsourced database scenarios. Moreover, we provide efficient and secure means of ensuring data integrity and authenticity while incurring minimal computational overhead. We provide techniques based on Message Authentication Codes (MACs) to detect malicious and/or unauthorized insertions, updates and deletions of data. Is this paper, we extend the work of [12], by enhancing the experimental evaluation, providing the algorithms for the proposed techniques and presenting a technique to provide *completeness* assurance of queries.

The remainder of this paper is divided into five sections. In Section II , we discuss related work. In Section III, we present techniques for providing data integrity and authenticity assurance. In Section IV, we analyse the performance impact of our proposed method and Section V presents our final considerations and future works.

II. RELATED WORK

The major part of integrity verification found in literature is based on authenticated structures. Namely, Merkle Hash Trees MHT [10] and Skip-Lists [11].

Li et al. [4] present the Merkle B-Tree (MB-Tree), where the B^+-tree of a relational table is extended with digest information as in an MHT. The MB-Tree is then used to provide proofs of correctness and completeness for posed queries to the server. Despite presenting an interesting idea and showing good results in their experiments, their approach suffers from a major drawback. To deploy this approach, the database server needs to be adapted as the B^+-tree needs to be extended to support an MHT. Such modifications may not

 ISBN: 978-1-61208-334-6

be feasible in real world environments, especially those that are already in use.

Di Battista and Palazzi [7] propose to implement an authenticated skip list into a relational table. They create a new table, called *security table*, which stores an authenticated skip list. The new table is then used to provide assurance of the authenticity and completeness of posed queries. This approach overcomes the requirement of a new DBMS, present in the previous approach. While only a new table is necessary within this approach, its implementation can be done as a plug-in to the DBMS. However, the experimental results are superficial. It is not clear what is the actual overhead in terms of each SQL operation. Moreover, their experiments show that the overhead increase as the database increases, while in our approach the overhead is constant in terms of the database size.

Miklau and Suciu [9] implement a hash tree into a relational table, providing integrity checks for the data owner. The data owner needs to securely store the root node of the tree. To verify the integrity, the clients need to rebuild the tree and compare the root node calculated and stored. If they match, the data was not tampered with. Despite using simple cryptographic functions, such as hash, the use of trees compromises the efficiency of their method. A tuple insert using their method is 10 times slower than a normal insert, while a query is executed 6 times slower. In our experiments, presented in section IV, we show that the naive implementation of our method is as good as their method.

E. Mykletun et al. [13] study the problem of providing *correctness* assurance of the data. Their work is most closely related to what we present in this paper. They present an approach for verifying data integrity, based on digital signatures. The client has a key pair and uses its private key to sign each tuple he/she sends to the server. When retrieving a tuple, the client uses the correspondent public key to verify the integrity of the retrieved tuple. This work was extended by Narasimha and Tsudik [14] to also provide proof of *completeness*.

The motivation of the authors to use digital signature is to allow integrity checking in multi-querier and multi-owner models. Therefore, for multi-querier and multi-owner models, their work is preferable. On the other hand, if the querier and the data owner are the same, our work can provide integrity assurance more efficiently. Moreover, our method can provide the same security level while consuming less of the servers resources. This is possible because to achieve the same security level, asymmetric keys are larger than symmetric keys. For example, for achieving the security level of a 2048 bit long asymmetric key, we need a 112 bit long symmetric key [15]. This reduces the amount of data required to control the integrity by a factor of 18, meaning that the data owner will be able to outsource more data.

Additionally, following a different approach, Xie et al. [16] proposes a probabilistic method to audit queries of outsourced databases. They insert a small quantity of fake tuples along with the real tuples of the database to control and audit the integrity of the system. Their method shows to be efficient, since it doesn't require any complex functions. However, their focus is on query integrity while in our work we are focused on the integrity of the data itself.

III. Providing integrity assurance for database content

To achieve a low cost method to provide integrity and authenticity, we propose to perform the cryptographic operations on the client side (application), using of Message Authentication Codes (MAC) [17], [18]. The implementation consists of adding a new column to each table. This new column stores the output of the MAC function applied to the concatenation ($||$) of the attributes (all columns, or a subset of them) of a row n, as shown in (1). The function also utilises a key k, which is only known by the application. The value of the MAC column is later used to verify integrity and authenticity.

$$MAC_n = MAC(k, Column1||...||Columni) \quad (1)$$

The use of a MAC function ensures the integrity of the INSERT and UPDATE operations. However, the table is still vulnerable to the unauthorized deletion of rows. To overcome this issue, we propose a new algorithm for linking sequential rows, called "Chained-MAC (CMAC)". The result of the CMAC is then stored into a new column. The value of this column, given a row n, a key k, and MAC_n as the MAC value of the row n, is calculated as shown in (2), where $\oplus$ denotes the exclusive OR operation (XOR).

$$CMAC_n = MAC(k, (MAC_{n-1} \oplus MAC_n)) \quad (2)$$

The use of CMAC provides an interesting property to the data stored in the table where it is used. When used, the CMAC links the rows in a way that an attacker cannot delete a row without being detected, since he does not have access to the secret key to produce a valid value to update the CMAC column of adjacent rows. Moreover, calculating the CMAC is very efficient, since we calculate only two MACs and a $\oplus$. Updating rows is also efficient. The CMAC is not a cascading operation, that is, it only needs to be updated when the MAC of a given row is updated. Figure 1 shown an example of a table with the MAC and CMAC columns. The circles represent the value of the MAC/CMAC and the arrows shows the MACs used to calculate a specific CMAC.

Despite linking adjacent rows, any subset of the first and last rows can be deleted without being detected. This is possible because the first row has no previous row and the last row does not have a subsequent row to be linked with. To overcome this issue, we propose changing the CMAC to a circular method. That is, for the first row, the $n-1$-th row to be considered will be the n-th row (i.e. the last row). With this change, if the last row is deleted, the integrity check will fail for the first row. Similarly, since the first row now has a predecessor, integrity checks can start at the first row (in the regular mode it would always start in the second row).

It is important to notice that the introduction of the CMAC brings a new requirement: the table must be ordered by some

 ISBN: 978-1-61208-334-6

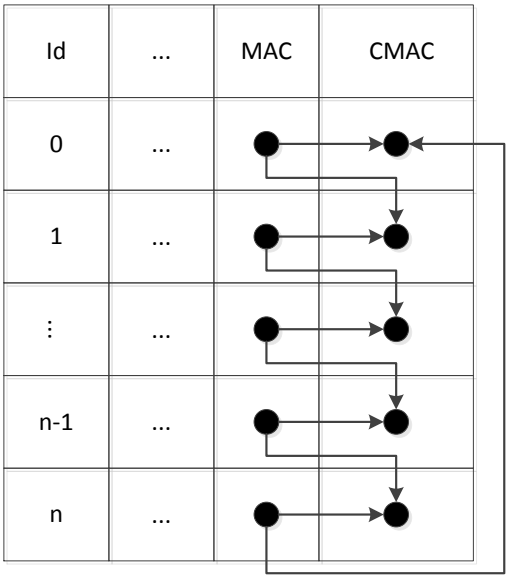

Fig. 1. Graphical representation of a table with the CMAC column

attribute. However, in real world scenarios, all tables have a primary key, and all the main DBMS orders the tables in terms of the primary key. Therefore, the requirement for a ordered table of the CMAC does not have a big impact to the deployment of our technique in real world scenarios.

A. Adding rows

The insertion of a new row into the database is straightforward. The client must calculate the MAC as shown in (1). The value of the CMAC column must be calculated using the MAC value of the previous row and the MAC value of the row being added, as shown in (2). Figure 2 shows the algorithm for the insertion of both MAC and CMAC values.

Input: A set T of values $t_0, \ldots, t_i$, following the schema of a table R and a key k, used to calculate the MAC.
Step 1: Calculate $MAC_{n+1} = MAC(k, t_0 || \ldots || t_i)$
Step 2: Calculate the CMAC
Step 2.1: Retrieve the MAC of the last row of R, denoted by MAC_n
Step 2.2: Calculate $CMAC_{n+1} = MAC(k, (MAC_n \oplus MAC_{n+1}))$
Step 3: Recalculate the CMAC of the first row (for the circular CMAC)
Step 3.1: Retrieve the MAC of the first row of R, denoted by MAC_1
Step 3.2: Calculate $CMAC_1 = MAC(k, (MAC_{n+1} \oplus MAC_1))$
Step 4: Insert the set T along with the calculated MAC_{n+1} , $CMAC_{n+1}$ and $CMAC_1$

Fig. 2. Algorithm for inserting a new row with the MAC and CMAC values

B. Updating rows

Updating rows is similar to the INSERT operation. The MAC and CMAC columns must be recalculated with the updated values. However, an extra step is necessary, which is to update the CMAC value of the next row, since the MAC of its previous row has been updated. Figure 3 shows the algorithm for updating a row.

Input: A set T of values $t_0, \ldots, t_i$, following the schema of a table R and a key k, used to calculate the MAC.
Step 1: Calculate $MAC_n = MAC(k, t_0 || \ldots || t_i)$
Step 2: Calculate the CMAC
Step 2.1: Retrieve the MAC of the previous row of R, denoted by MAC_{n-1}
Step 2.2: Calculate $CMAC_n = MAC(k, (MAC_{n-1} \oplus MAC_n))$
Step 3: Update the set T along with the calculated MAC_n and $CMAC_n$
Step 4: Calculate the CMAC of the $n+1$th-row. If the nth-row is the last row in the table, then the $n+1$th-row to be considered will be the first row of the table.
Step 4.1: Retrieve the MAC of the $n+1$th-row row of R, denoted by MAC_{n+1}
Step 4.2: Calculate $CMAC_{n+1} = MAC(k, (MAC_n \oplus MAC_{n+1}))$
Step 4.3: Update the calculated $CMAC_{n+1}$

Fig. 3. Algorithm for updating a row with the MAC and CMAC values

C. Deleting rows

To delete a row of a table that uses only the MAC column, no additional actions are needed. The reason for this is that, by using only MAC, it is not possible to check the integrity of a table against unauthorised row deletion. When using a CMAC column, the application needs to recalculate the value of the next row by referencing the previous row. Figure 4 shows the algorithm for deleting a row.

Input: A set T of values $t_0, \ldots, t_i$, following the schema of a table R and a key k, used to calculate the MAC.
Step 1: Delete T
Step 2: Calculate the CMAC of the $n+1$-th row. If the n-th row is the last row, then the $n+1$-th row to be considered is the first row of R.
Step 2.1: Retrieve the MAC of the $n+1$-rh row of R, MAC_{n+1} and the MAC of the previous row, MAC_{n-1}
Step 2.2: Calculate $CMAC_{n+1} = MAC(k, (MAC_{n-1} \oplus MAC_{n+1}))$
Step 2.3: Update the calculated MAC, $CMAC_{n+1}$

Fig. 4. Algorithm for deleting a row with the MAC and CMAC values

D. Verifying the integrity of a table

Data integrity can be provided in different levels of granularity. We can perform integrity checks of a table (entire

relation), a row (a record or a tuple of the table) or a column (an attribute of the relation). Providing integrity checks at the table level, implies that every row of the table must have the MAC column filled and its value must be calculated based on every attribute of the row. Providing integrity checks of a single row implies that that specific row must have the MAC column filled with the result of the MAC function applied on the concatenation of all attributes. Finally, providing integrity checks at the attribute level implies that the MAC function must be applied to a specific set of attributes only.

To verify the integrity of a row with the MAC column, the application must calculate the MAC of that row and compare it with the value of the MAC column. The row can be considered as not modified if the calculated MAC is equal to the stored MAC. Applying this comparison to each row of a table will ensure the integrity of this table against insertion and modification attacks. As stated earlier, the use of the MAC does not provide a means to verify the integrity of a table against unauthorized deletions. In this case, the CMAC column should be used. To verify the integrity of a table with the CMAC column, the application must check the integrity of each pair of sequential registries of the table. That is, a Table T has not been (unauthorized) modified if:

$$\forall t_{n-1}, t_n \in T : t_n.CMAC = CMAC(k, t_{n-1}, t_n) \quad (3)$$

E. Verifying the completeness of queries

The CMAC mechanism can also be used for verifying the *completeness* of simple range queries. This is possible due to the catenation of adjacent rows. If the data owner does not trust the DMBS software on the server and therefore needs a guarantee that the server is not omitting valid results for posed queries, the client should proceed as shown in Figure 5.

Input: A query Q
Step 1: Pose Q to the server, which will return a set T of values $t_i, \ldots, t_j$, where $i <= j$
Step 2: Retrieve the rows t_{i-1} and t_{j+1}
Step 3: Verify the integrity of the values $t_{i-1}, t_i, \ldots, t_j, t_{j+1}$, as described in section III-D

Fig. 5. Algorithm for verifying the *completeness* of a query

If the result of the integrity check is positive, then we know that no intermediate result has been omitted by the server. However, just verifying the integrity does not guarantee that a value has not been omitted at all. If the server omits the values in the edge, the integrity check will still pass. To guarantee that the values in the edge have not been omitted, the client needs to check whether the edge tuples are the same tuples retrieved in step 2 or not. If these tuples are the same and the integrity check passed, then all values satisfying the query have been returned by the server.

IV. Performance analysis

To assess the efficiency of our techniques we implemented a tool to evaluate the performance of using a Keyed-Hash Message Authentication Code (HMAC), as the MAC function, and CMAC. The prototype was implemented using the C programming language and the OpenSSL library. The DBMS used was MySQL database and the experiments were performed in a machine running both MySQL server and client application. The machine had Intel Core 2 Quad CPU Q8400 with 4Mb cache, at 2.66GHz, 4GB RAM 800Mz, and 320Gb disk, SATAII, 16Mb cache, 7200RPM, running an Ubuntu 11.04 32-bit operating system with OpenSSL 0.9.8d and MySQL 5.1. Additionally, we used the SHA-1 hash function to calculate the HMAC with a 256-bit long key and disabled the cache of the MySQL.

We considered different scenarios to evaluate the performance of the proposed techniques. For each scenario, we executed the workload a thousand times over a table with 10 thousand tuples of random values. All the results shown below are the average of these executions. In all scenarios, we focus on evaluating the amount of time spent on the operations of INSERT, UPDATE, DELETE and SELECT, performed under four distinct conditions:

1) Without security mechanisms;
2) Using HMAC only;
3) Using both HMAC and CMAC;
4) Using both HMAC and CMAC in the circular mode.

Insert

In the first scenario, we focused on measuring and comparing the execution times for the INSERT operation under each specified condition. The results (as we can see in Figure 6) show that the baseline took 42,3 μs, while the HMAC took 47 μs, 90% of which is spent on the server side and 10% on the client side. The scenario with the use of CMAC executed in 118,3 μs, with 91% of the time spent on the server and 9% on the client. The CMAC in the circular mode executed in 331,7 μs, where 72% is executed by the server and 28% by the client.

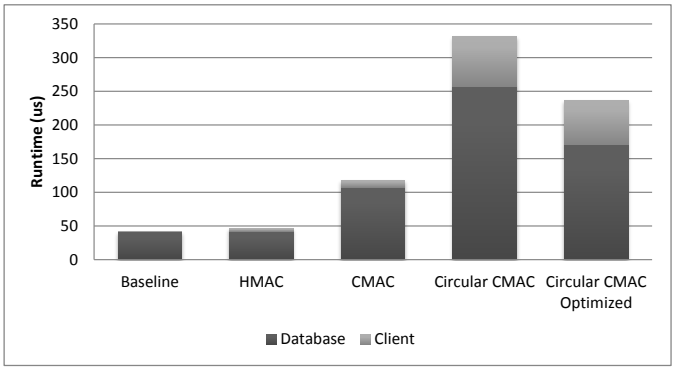

Fig. 6. Comparison of execution time of the INSERT operation

The CMAC in the circular mode can be optimized if the client store a small amount of data. The major reason for the difference between the the regular mode and the circular mode of the CMAC is that in the circular mode we need to retrieve and update additional rows. If the client stores the first row

locally, we eliminate one query, reducing the execution time from 331,7 μs to 236,5 μs.

Update

In the second scenario, we focused on measuring and comparing the execution times for the UPDATE operation under each specified condition. The results (shown in Figure 7) show that the baseline took 127,6 μs, while the HMAC took 134 μs, 95% of which is spent on the server side and 5% on the client side. The CMAC (both in regular and circular mode) executed in 381,9 μs, with 80% of the time spent on the server and 20% on the client. The reason that the execution time for the CMAC in the regular and circular mode are the same is because they execute the exact same operations.

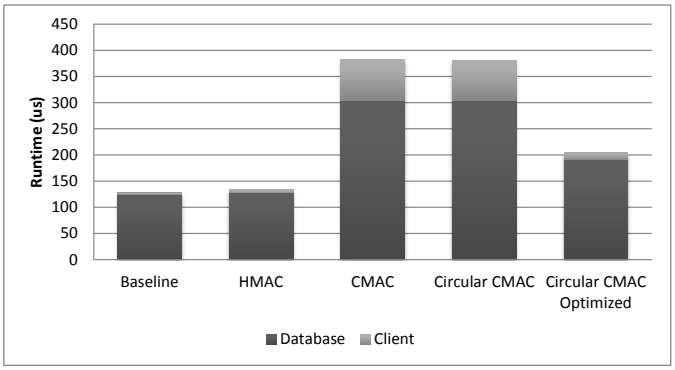

Fig. 7. Comparison of execution time of the UPDATE operation

We can also optimize the CMAC for the UPDATE operation if we consider that some values are available on the client side in the moment of the operation. In this case, when updating a row n, we need the MAC and CMAC of the $n+1$-th and the $n-1$-th rows (for example, these values could have been retrieved when the client retrieved the n-th row). If these rows are available on the client side in the moment of the update, the execution time is 204,5 μs.

Delete

In the third scenario, we focused on measuring and comparing the execution times for the DELETE operation under each specified condition. The baseline executed in 51 μs and when using the HMAC to delete a row, there is no additional cost since there is no extra operations to be performed (as shown in Tables I and II). On the other hand, the CMAC (both in regular and circular mode) executed in 186,5 μs, with 96% of the time spent on the server and 4% on the client, as we can see in Figure 8. As we have shown for the UPDATE operation, the CMAC in the regular and circular mode have the exact same operations and therefore the overhead is the same.

We can use the same idea presented for the UPDATE operation to improve the efficiency of the CMAC. In the naive implementation, before deleting a row n, we execute a select query to retrieve the $n+1$-th and the $n-1$-th rows. Considering that these rows are available on the client side in the moment of the delete, the execution time is reduced from 186,5 μs to 105,2 μs.

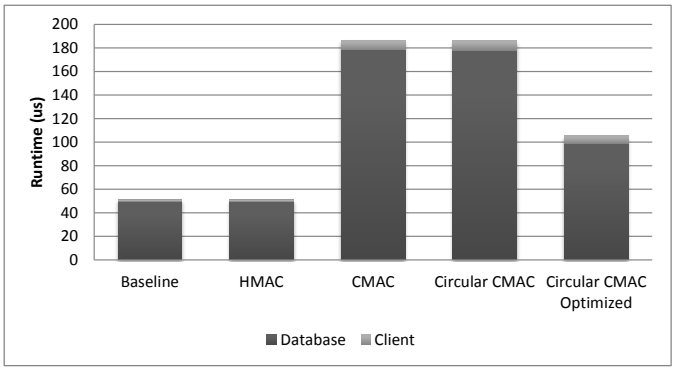

Fig. 8. Comparison of execution time of the DELETE operation

Select

Finally, in the last scenario, we focused on measuring and comparing the execution times to check the integrity during the SELECT operation under each specified condition. A SELECT query, without verifying the integrity (the baseline) took 18,4 μs. To verify the integrity of the HMAC the client needs to recalculate the HMAC and compare it to the one retrieved from the server. This operation executed in 22,5 μs, due to the calculation of the HMAC. When using the CMAC, the client needs to retrieve the HMAC of the previous row and recalculate both the HMAC and CMAC. These extra operations increase the execution time to 54 μs, as we can see in Figure 9. However, if we consider that the previous row is available on the client side, the execution time is reduced to 27,6 μs (for example, the client retrieved the n-th and $n-1$th rows in a single query).

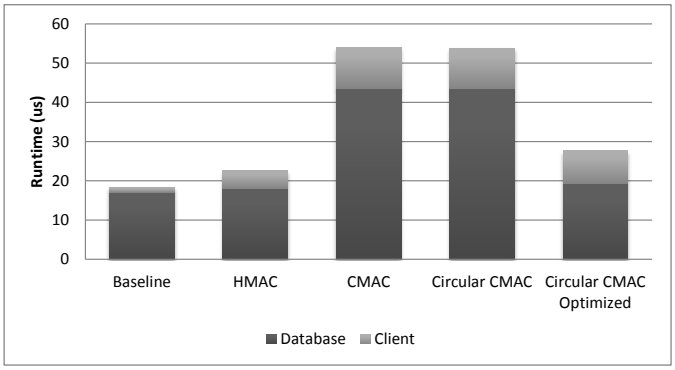

Fig. 9. Comparison of execution time of the SELECT operation

Summarizing

As we could see in the results, the impact of using each method is affected by two main factors: i) the number of sql operations; ii) the number of cryptographic functions performed. The number of sql operations performed by every type of action (ex: an update when using CHMAC requires two SQL operations) clearly has a bigger impact on the performance. The cryptographic-only operation have shown very low impacts. The table I shows the number of SQL

operations each method requires. The table II shows the number of cryptographic functions each method requires.

Although the cost of the SQL operations are 2-7 times greater with our method, it is more efficient than previous work. E. Mykletun et al. [13] and Narasimha and Tsudik [14] uses asymmetric cryptography, which is 1,000-10,000 times slower than simple MAC functions. In terms of the storage, our method generates less data to be stored in the database. For example, considering an RSA 2048-bit long key and the SHA-1 function, which are the values recommended by NIST [15] for 2014, our method generates approximately 13 times less data to control the integrity of the database.

Additionally, when comparing to the approaches based on authenticated structures [7], [8], [9], [4], our method has a lower complexity. That is, the cost of our method is constant to the database size ($O(1)$) while the approaches based on authenticated structures are usually logarithmic ($O(\log n)$). Therefore, for larger databases our method is more efficient.

Is is important to notice that the DBMS cache was disabled while running the tests. In a real environment, with the cache enabled, the retrieval of previous row necessary to calculate the CMAC will be cached with a high probability, reducing the total cost of each SQL operation.

V. Final remarks

This paper proposes secure and efficient methods for providing integrity and authenticity for relational database systems. Our methods focus on strategies for detecting unauthorised actions (insertions, deletions and updates) from a vulnerable database server.

Prior work either requires modifications in the database implementation or uses inefficient cryptographic techniques (for example, public key cryptographic). The requirement of modifying the core of a database system makes the deployment of these methods difficult in real world scenarios. Thus, one significant advantage of our method is that it is DBMS-independent and can be easily deployed in existing environments. Another advantage of our method is that we focused on using a more simple and efficient cryptographic algorithm to provide the integrity checks.

The performance requirements for each of our methods were presented and alternatives to minimise their costs and its consequences were discussed. Finally, we believe that the transparency and independency of our method makes it easily deployable and compatible with real world demands.

As a future work, we would like to address the roll-back attack. A roll-back attack is characterized when the attacker restores the database to a previous valid state. With this attack, the attacker can delete rows without being noticed, for example. Prior work, as well as this paper, are vulnerable to such attacks. Another interesting matter of research is to address the actions to be taken in case of a detected attack. We consider that the ideal solution is to restore the database to a valid state before the attack.

References

[1] P. Samarati and S. D. C. di Vimercati, "Data protection in outsourcing scenarios: Issues and directions," in Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, ser. ASIACCS '10. New York, NY, USA: ACM, 2010, pp. 1–14. [Online]. Available: http://doi.acm.org/10.1145/1755688.1755690

[2] T. K. Dang, "Ensuring correctness, completeness, and freshness for outsourced tree-indexed data," Inf. Resour. Manage. J., vol. 21, no. 1, Jan. 2008, pp. 59–76. [Online]. Available: http://dx.doi.org/10.4018/irmj.2008010104

[3] I. Kamel, "A schema for protecting the integrity of databases," Computers & Security, vol. 28, no. 7, 2009, pp. 698–709.

[4] F. Li, M. Hadjieleftheriou, G. Kollios, and L. Reyzin, "Dynamic authenticated index structures for outsourced databases," in Proceedings of the 2006 ACM SIGMOD international conference on Management of data, ser. SIGMOD '06. New York, NY, USA: ACM, 2006, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/1142473.1142488

[5] T. Aditya, P. Baruah, and R. Mukkamala, "Employing bloom filters for enforcing integrity of outsourced databases in cloud environments," in Advances in Computing and Communications, ser. Communications in Computer and Information Science, A. Abraham, J. Lloret Mauri, J. Buford, J. Suzuki, and S. Thampi, Eds. Springer Berlin Heidelberg, 2011, vol. 190, pp. 446–460. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-22709-7_44

[6] T. Aditya, P. K. Baruah, and R. Mukkamala, "Space-efficient bloom filters for enforcing integrity of outsourced data in cloud environments," in Proceedings of the 2011 IEEE 4th International Conference on Cloud Computing, ser. CLOUD '11. Washington, DC, USA: IEEE Computer Society, 2011, pp. 292–299. [Online]. Available: http://dx.doi.org/10.1109/CLOUD.2011.40

[7] G. Di Battista and B. Palazzi, "Authenticated relational tables and authenticated skip lists," in Proceedings of the 21st annual IFIP WG 11.3 working conference on Data and applications security. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 31–46. [Online]. Available: http://dl.acm.org/citation.cfm?id=1770560.1770564

[8] A. Heitzmann, B. Palazzi, C. Papamanthou, and R. Tamassia, "Efficient integrity checking of untrusted network storage," in Proceedings of the 4th ACM international workshop on Storage security and survivability, ser. StorageSS '08. New York, NY, USA: ACM, 2008, pp. 43–54. [Online]. Available: http://doi.acm.org/10.1145/1456469.1456479

[9] G. Miklau and D. Suciu, "Implementing a tamper-evident database system," in Proceedings of the 10th Asian Computing Science conference on Advances in computer science: data management on the web, ser. ASIAN'05. Berlin, Heidelberg: Springer-Verlag, 2005, pp. 28–48. [Online]. Available: http://dl.acm.org/citation.cfm?id=2074944.2074951

[10] R. C. Merkle, "A certified digital signature," in CRYPTO, ser. Lecture Notes in Computer Science, G. Brassard, Ed., vol. 435. Springer, 1989, pp. 218–238.

[11] W. Pugh, "Skip lists: a probabilistic alternative to balanced trees," Commun. ACM, vol. 33, no. 6, Jun. 1990, pp. 668–676. [Online]. Available: http://doi.acm.org/10.1145/78973.78977

[12] R. d. S. M. Anderson Luiz Silvério and R. F. Custódio, "Efficient integrity checking for untrusted database systems," in Proceedings of the 28th Brazilian Symposium on Databases, ser. WTDBD'13. Sociedade Brasileira de Computação, 2013, pp. 36–42.

[13] E. Mykletun, M. Narasimha, and G. Tsudik, "Authentication and integrity in outsourced databases," ACM Transactions on Storage, vol. 2, no. 2, May 2006, pp. 107–138. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1149976.1149977

[14] M. Narasimha and G. Tsudik, "Dsac: integrity for outsourced databases with signature aggregation and chaining," in Proceedings of the 14th ACM international conference on Information and knowledge management, ser. CIKM '05. New York, NY, USA: ACM, 2005, pp. 235–236. [Online]. Available: http://doi.acm.org/10.1145/1099554.1099604

[15] E. Barker, W. Barker, W. Burr, W. Polk, and M. Smid, "Recommendation for key management - pat1: General (revision 3)," National Institute of Standards and Technology, NIST Special Publication 800-57, Jul 2012. [Online]. Available: http://csrc.nist.gov/publications/nistpubs/800-57/sp800-57_part1_rev3_general.pdf

 ISBN: 978-1-61208-334-6

TABLE I
NUMBER OF SQL OPERATIONS PERFORMED IN EACH METHOD

	No protection	MAC	CMAC	Circular CMAC	CMAC Optimized
Insert	1	1	2	3	2
Update	2	2	4	5	2
Delete	1	1	3	3	2
Select	1	1	2	2	1

TABLE II
NUMBER OF CRYPTOGRAPHIC OPERATIONS PERFORMED IN EACH METHOD

	No protection	MAC	CMAC	Circular CMAC	CMAC Optimized
Insert	0	1	2	3	3
Update	0	1	3	3	3
Delete	0	1	1	1	1
Select	0	1	2	2	2

[16] M. Xie, H. Wang, and J. Yin, “Integrity auditing of outsourced data,” Very large data bases, 2007, pp. 782–793. [Online]. Available: http://dl.acm.org/citation.cfm?id=1325940

[17] M. Bellare, R. Canetti, and H. Krawczyk, “Keying hash functions for message authentication,” in Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology, ser. CRYPTO ’96. London, UK, UK: Springer-Verlag, 1996, pp. 1–15. [Online]. Available: http://dl.acm.org/citation.cfm?id=646761.706031

[18] H. Krawczyk, M. Bellare, and R. Canetti, “HMAC: Keyed-Hashing for Message Authentication,” RFC 2104 (Informational), Internet Engineering Task Force, Feb. 1997, updated by RFC 6151. [Online]. Available: http://www.ietf.org/rfc/rfc2104.txt

 ISBN: 978-1-61208-334-6